Lasso parallèle

Sébastien COUBE et Benjamin PHAN

I Introduction

Nous avons durant notre projet étudié un article[1] d'Ilya Trofimov et Alexander Genkin décrivant une méthode permettant d'implémenter une régression avec pénalité L_1 (a.k.a. régression Lasso) de manière distribuée. C'est une méthode de *coordinate descent*. Nous avons implémenté l'algorithme en Python et nous avons en particulier utilisé le package *threading* en ce qui concerne la parallélisation et nous l'avons essayé sur un jeu de données qu'on a simulé.

II Description du problème

Nous nous plaçons dans un problème de classification binaire. La variable d'intérêt Y peut prendre deux valeurs : 1 et -1. On pose donc un modèle logistique :

Probabilité que $y = 1$ sachant x

$$P(y = 1|x) = \frac{1}{1 + \exp(-\beta^T x)} \tag{1}$$

Log-vraisemblance négative

$$L(\beta) = \sum_{i=1}^{n} \log(1 + \exp(-y_i \beta^T x)) \tag{2}$$

Si on se place dans le cadre d'une régression Lasso, on cherche le paramètre β^* tel que

$$\beta^* = \arg\min_{\beta} f(\beta) := L(\beta) + \lambda \|\beta\|_1 \tag{3}$$

avec λ le parmètre de pénalité de notre régression.
On note p la dimension de β et n le nombre d'observations (y_i, x_i)

III L'algorithme

En ajoutant la pénalité L_1 nous avons une fonction objectif qui n'est pas dérivable en tout point, ce qui est un problème pour des méthodes comme la descente de gradient. On pose alors l'approximation suivante :

$$L_q(\beta, \Delta\beta) = L(\beta) + \nabla L(\beta)^T \Delta\beta + \frac{1}{2} \Delta\beta^T \nabla^2 L(\beta) \Delta\beta \tag{4}$$

$$= \frac{1}{2} \sum_{i=1}^{n} w_i (z_i - \Delta\beta^T x_i)^2 + C(\beta) \tag{5}$$

Avec $C(\beta)$ une constante indépendante de $\Delta\beta$ et

$$p(x_i) = \frac{1}{1 + e^{-\beta^T x_i}}$$
$$w_i = p(x_i)(1 - p(x_i))$$
$$z_i = \frac{(y_i + 1)/2 - p(x_i)}{w_i}$$

[1] `https://arxiv.org/pdf/1411.6520v1.pdf`

Le principe de l'algorithme est de modifier de manière itérative β en trouvant le $\Delta\beta$ qui minimise $L_q(\beta, \Delta\beta) + \lambda \|\beta + \Delta\beta\|_1$

Le $\Delta\beta^*$ optimal est défini par :

$$\Delta\beta_j^* = \frac{T(\sum_{i=1}^n w_i x_{ij} q_i, \lambda)}{\sum_{i=1}^n w_i x_{ij}^2} \tag{6}$$

avec

$$T(x, a) = sign(x) max(|x| - a, 0)$$
$$q_i = z_i - \Delta\beta^T x_i + (\beta_j + \Delta\beta_j) x_{ij}$$

Ceci nous permet donc de calculer séparément les coordonnées de $\Delta\beta^*$ ce qui est favorable à une parallélisation de cette étape. En effet on sépare les coordonnées en M blocs ; on crée une partition :

$$\bigcup_{m=1}^{M} S_m = \{1, \ldots, p\}$$

On distribue alors le calcul à M machines s'occupant chacune des coordonnées correspondant à un S_m et on somme ensuite les résultats.

Une fois le $\Delta\beta^*$ obtenu, on procède au *linesearch*. Le $\Delta\beta^*$ nous donne une direction dans laquelle modifier β pour optimiser notre fonction objectif et le *linesearch* nous indique le pas à utiliser. Cette étape consiste à trouver un réel α entre 0 et 1 pour ensuite mettre à jour β par : $\beta t + 1 = \beta_t + \alpha\Delta\beta_t^*$. Il s'agit en fait d'appliquer l'algorithme suivant :

- Si $\alpha = 1$ permet de faire décroître *suffisamment* la fonction objectif : choisir $\alpha = 1$

$$\alpha_{init} = \arg \min_{\delta < \alpha \leq 1} f(\beta + \alpha\Delta\beta),\ \delta > 0$$

- Règle d'Armijo :

$$\alpha = \arg\max \left\{ \alpha_{init} b^j : f(\beta + \alpha\Delta\beta) \leq f(\beta) + \alpha\sigma D,\ j \in \mathbb{N} \right\}$$

avec $0 < b < 1,\ 0 < \sigma < 1,\ 0 \leq \gamma < 1$ et

$$D = \nabla L(\beta)^T \Delta\beta + \gamma \Delta\beta^T H \Delta\beta + \lambda(\|\beta + \Delta\beta\|_1 - \|\beta\|_1)$$

H étant la hessienne $\nabla^2 L(\beta)$. L'article propose de remplacer la hessienne par une version diagonale par blocs et d'y ajouter une indicatrice multipliée par un coefficient très faible pour des raisons computationnelles.
Choisir un coefficient $\alpha < 1$ peut nuire à la propriété du Lasso consistant à annuler certains coefficients de l'estimateur. C'est pourquoi on choisit $\alpha = 1$ si cela permet de réduire suffisamment la fonction objectif. C'est un paramètre qui est à notre charge de fixer. On a estimé qu'une baisse de 5% était suffisante.

Une fois α obtenu, on met à jour β et on recommence jusqu'à convergence de l'estimateur.

IV Notre implémentation

IV.1 Simulation de données

Nous avons simulés des données afin d'essayer l'algorithme décrit dans l'article. Nous avons d'abord simulé un paramètre $\beta \in \mathbb{R}^p$ avec $p = 1000$. Nous avons simulé ce paramètre de sorte à ce qu'il y ait beaucoup de coefficients nuls et qu'il y ait donc intérêt à faire une régression Lasso, qui est réputée pour être parcimonieuse dans la sélection des variables. Nous avons simulé chaque coefficient de β

de manière indépendante : avec une probabilité de 95% le coefficient est nul, sinon il suit une loi de Laplace(0,1).
Ensuite nous avons simulé un échantillon de $n = 200$ variables explicatives $(x_i)_{0 \leq i < n}$ iid de $\mathbb{R}^{150}$. Chaque coefficient suit une loi Normale centrée réduite indépendante des autres coefficients à l'exception d'une variable constante afin d'autoriser une constante non nulle dans le modèle.
Enfin nous avons simulé la variable d'intérêt Y de la manière suivante $y_i = sign(\beta^T x_i + \epsilon_i)$ avec $\epsilon_i \sim \mathcal{N}(0,1)$ iid. Nous avons alors un jeu de données complet adapté à notre problème.

IV.2 Implémentation de l'algorithme

Nous avons essayé de séparer les différentes étapes en plusieurs petites fonctions ce qui permet une plus grande clarté. Nous avons d'abord créé des fonctions permettant de faire des calculs intermédiaires : le calcul des grandeurs $p(x_i), w_i, z_i$ ou encore la fonction $T(x, a)$

La première difficulté rencontrée est le calcul des $\Delta\beta_j^*$. En effet, l'article donne une formule donnant les $\Delta\beta_j^*$ mais elle fait intervenir q_i qui eux-mêmes dépendent de $\Delta\beta$. Il n'est alors pas très clair comment il faut procéder pour obtenir les $\Delta\beta_j^*$. Ce qui est sûr c'est que $\Delta\beta_j^*$ est un point fixe quand on calcule q_i à l'aide de $\Delta\beta^*$. Puisqu'il est demandé de minimiser $L_q(\beta, \Delta\beta^m) + \lambda \left\|\beta + \Delta\beta\right\|_1$ sur $\Delta\beta^m$, on initialise $\Delta\beta$ à 0 dans chaque machine puis on fait une boucle en appliquant l'équation 6 jusqu'à ce que $\Delta\beta^m$ ne bouge (presque) plus. À noter qu'au sein de la même machine, on prend en compte les premières coordonnées calculées pour calculer les suivantes. L'article dit qu'il faut faire optimiser sur chaque machine séparément en fixant à 0 les $\Delta\beta^{m'},\ m' \neq m$ et ensuite faire la somme de tous les résultats. Cependant on a estimé que ce n'était pas le plus pertinent. En effet, si l'une des machines met du temps à converger alors toutes les autres doivent l'attendre. À la place on a pensé que c'était mieux d'initialiser $\Delta\beta$ à 0 puis d'appliquer une fois l'équation 6 pour chaque machine, sommer le résultat puis ré-appliquer l'équation 6 sur chaque machine et sommer à nouveau jusqu'à convergence. Ainsi, toutes les machines sont mises à contribution à peu près tout le temps.

Nous avons ici affaire à un *model parallelism* ; toutes les machines travaillent sur les mêmes données mais chaque machine se charge de calculer des paramètres différents. On a alors eu recours à plusieurs threads ce qui permet de travailler sur les mêmes objets en parallèle. En particulier, les threads modifient tous la variable $\Delta\beta$. On a donc besoin d'un verrou (*lock*) pour s'assurer qu'un seul thread ne modifie la variable à la fois. Sinon il peut il y avoir des problèmes d'écritures concurrentes : deux threads prennent la variable en même temps, la modifient et puis renvoie leur résultats. On obtient alors le résultat de l'un des deux sans qu'il ait pris en compte le résultat de l'autre qui aura alors été omis. C'est pourquoi l'utilisation du verrou est indispensable.

Une fois $\Delta\beta^*$ trouvé, il reste à faire l'étape du *linesearch*. Il faut alors fixer quelques paramètres, qu'on a défini par défaut :

- baisse suffisante pour choisir $\alpha = 1$: 5%
- $b = 0,95$
- $\sigma = 0,5$
- $\gamma = 0,5$
- $\delta = 0,2$

Pour appliquer l'algorithme de *linesearch*, nous avons eu besoin de créer une fonction calculant la valeur de la fonction objectif f : la log vraisemblance pénalisée (équation 3). On trouve alors le α_{init} qui minimise $f(\beta + \alpha\Delta\beta^*)$ par un algorithme d'optimisation sous contraintes du package *scipy*. Il faut ensuite appliquer la règle d'Armijo. On teste si la condition est vérifiée en commençant par $j = 0$ et en incrémentant j tant qu'elle ne l'est pas et on prend la première valeur qui satisfait la condition puisque

la suite $(\alpha_{init} b^j)_j$ est décroissante. Pour savoir si la condition est satisfaite, on a besoin de calculer D pour lequel on a besoin de calculer $\nabla L(\beta)$ ainsi que H. On a :

$$L(\beta) = \sum_{k=1}^{n} \log(1 + \exp(-y_k \beta^T x_k)) = \sum_{k=1}^{n} \log(1 + \exp(-y_k \sum_{i=1}^{p} \beta_i x_k^{(i)}))$$

Donc en dérivant on obtient :

$$\frac{\partial L(\beta)}{\partial x^{(i)}} = \sum_{k=1}^{n} \frac{-y_k \beta_i \exp(-y_k \beta^T x_k)}{1 + \exp(-y_k \beta^T x_k)}$$
$$\frac{\partial^2 L(\beta)}{\partial x^{(i)} \partial x^{(j)}} = \sum_{k=1}^{n} \frac{\beta_i \beta_j \exp(-y_k \beta^T x_k)}{(1 + \exp(-y_k \beta^T x_k))^2}$$

On remplace la hessienne par sa version diagonale par bloc et en ajoutant 10^{-10} à la diagonale comme proposé dans l'article. De plus, le fait de calculer seulement quelques blocs de la hessienne est adapté à la parallélisation. Nous avons alors fait une parallélisation similaire à celle pour le calcule des $\Delta\beta^m$: on utilise plusieurs threads qui travaillent sur les mêmes données et chaque thread calcule un bloc de la hessienne. Enfin, quand un thread a fini de calculer, il essaie de modifier la hessienne, ce pourquoi il a besoin d'acquérir un verrou. On a alors tous les éléments pour appliquer le *linesearch*, ce qui nous donne un coefficient α.

Ensuite on peut mettre à jour β en lui ajoutant $\alpha\Delta\beta$. Enfin, on fait une boucle pour tout recommencer tant que le critère de convergence n'est pas satisfait. Le critère de convergence est de s'arrêter si $\|\alpha\Delta\beta\|_1$ est plus petit qu'un seuil, qu'on a fixé par défaut à $\sqrt{p}.10^{-4}$, ou bien si la boucle a été exécutée trop de fois, qu'on a fixé par défaut à 10^2. La limite du nombre d'itérations sert à ne pas avoir un algorithme qui tourne indéfiniment dans le cas où il divergerait pour quelconque raison théorique ou erreur de programmation.

V Résultats

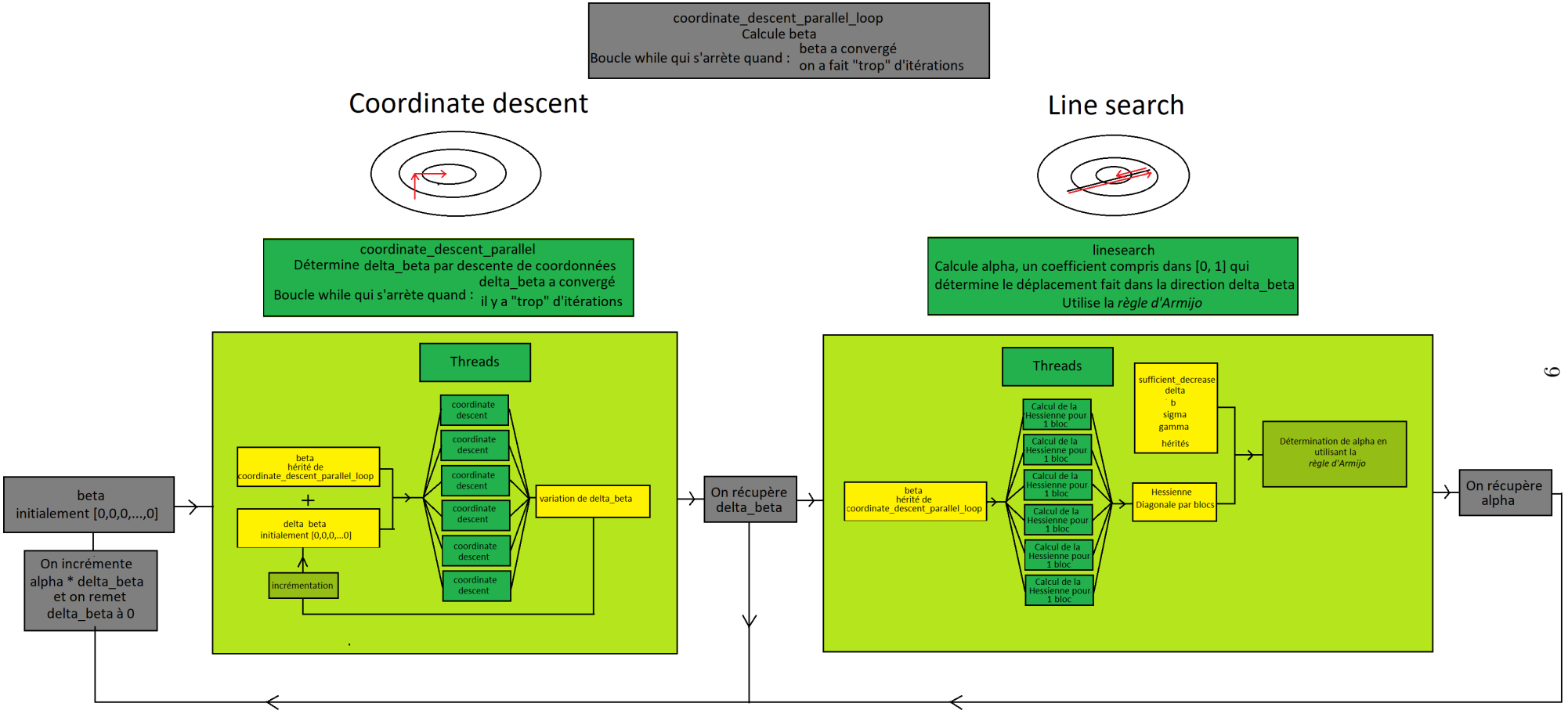
Nous avons essayé notre algorithme sur notre base de données avec différentes valeurs de λ et nous avons comparé les résultats de prédiction sur un jeu de données test qui n'a pas servi à la régression et qui a été simulé de la même manière que l'original.

On a comparé la vitesse de l'algorithme lorsqu'on a 5 threads et lorsqu'on a qu'un seul sur une base de 200 observations de 1000 variables. Celui avec 5 threads a mis 48 secondes alors que celui avec 1 thread a mis 46 secondes. Il est surprenant que le calcul prenne plus de temps avec plus de threads. On pourrait s'attendre naïvement que le temps soit divisé par 5 avec 5 threads mais tout l'algorithme n'est pas parallélisé et il y a donc des parties qui ne profitent pas du nombre de threads. De plus, plus il y a de threads plus il y a de coûts de communication entre les différents threads. En l'occurrence, plus il y a de threads et plus un thread risque d'attendre lorsqu'il essaie d'acquérir un verrou. Par ailleurs certaines opérations vectorisées sont optimisées dans les packages tels que *pandas* ou *numpy* en faisant appel à d'autres langages de programmation ; il peut alors être néfaste pour la performance d'essayer de découper ces opérations à la main. Ceci joue dans le sens où multiplier le nombre de threads par X ne divise pas le temps par X mais par moins. En revanche, il y a un autre effet qui joue dans l'autre sens : lorsqu'on a moins de threads, on découpe moins la matrice hessienne ce qui fait qu'il y a plus de coefficients à calculer. Cependant on remarque que le nombre de fois que la boucle doit être itérée pour atteindre la convergence peut différer avec le nombre de threads. Notamment, la hessienne a été calculée entièrement lorsqu'on a qu'un seul thread ce qui donne vraisemblablement une mise à jour de β plus précise ce qui peut réduire le nombre d'itération.
En particulier, c'est vraiment le cas pour le calcul de la Hessienne. Nous nous sommes prêtés au jeu de coder un calcul par bloc (car la Hessienne est bloc-diagonale), mais sur une matrice de dimension 1000*1000, un calcul en un seul thread est plus rapide qu'en 5 threads.
Pour tester les capacités de prédiction de l'algorithme, on simule des données de test et on tente de les

prédire.
Nous avons essayé l'algorithme avec différentes valeurs de λ. Il diverge pour de petites valeurs de λ. C'est dommage, car ce seraient justement de petites valeurs pour λ qui nous permettraient de trouver un paramètre estimé avec une norme L1 proche de celle du paramètre réel. Cependant, même avec une forte pénalité, le prédicteur obtenu obtient des scores significativement meilleurs que ceux d'un prédicteur aléatoire : avec 200 individus et 1000 variables, il se trompe dans 30 pourcents des cas environ. C'est cependant beaucoup moins bien qu'en utilisant les vrais paramètres, qui donnent environ 5 pourcents d'erreur.
Nous avons également essayé la régression Lasso implémentée dans le package *sklearn*. Elle est beaucoup plus rapide que notre algorithme, ce à quoi on pouvait s'y attendre puisque le package est optimisé de sorte à ce qu'il tourne rapidement et passe par des langages bas niveau ce qui permet d'exécuter plus rapidement les algorithmes.

VI Conclusion

Nous avons pu implémenter l'algorithme décrit dans l'article et gérer sa parallélisation à l'aide de plusieurs *threads*. L'algorithme a l'air de fonctionner convenablement pour des valeurs suffisamment grandes de λ (≥ 10). Pour des valeurs très grandes ($\lambda \geq 50$), on obtient, conformément à la théorie, un estimateur $\hat{\beta}$ nul. Cependant, les performances de l'algorithme ne sont pas aussi bonnes que ce que l'on pourrait espérer (notamment pour les faibles valeurs de λ) aussi bien dans les résultats que dans la vitesse d'exécution. Il est intéressant de coder cet algorithme à partir de rien (ou presque) mais dans un cadre pratique où on devrait faire une régression Lasso, il semble beaucoup plus raisonnable de se servir de packages prévus à cet effet comme *sklearn* qui sont reconnus pour la qualité de leurs résultats et leur rapidité.


coordinate_descent_parallel_loop
Calcule beta
Boucle while qui s'arrête quand : beta a convergé
on a fait "trop" d'itérations
Coordinate descent
Line search
coordinate_descent_parallel
Détermine delta_beta par descente de coordonnées
Boucle while qui s'arrête quand : delta_beta a convergé
il y a "trop" d'itérations
linesearch
Calcule alpha, un coefficient compris dans [0, 1] qui
détermine le déplacement fait dans la direction delta_beta
Utilise la règle d'Armijo
Threads
coordinate descent
beta hérité de coordinate_descent_parallel_loop
delta beta initialement [0,0,0,...0]
incrémentation
variation de delta_beta
beta initialement [0,0,0,...,0]
On incrémente alpha * delta_beta et on remet delta_beta à 0
On récupère delta_beta
Threads
Calcul de la Hessienne pour 1 bloc
sufficient_decrease
delta
b
sigma
gamma
hérités
Hessienne Diagonale par blocs
Détermination de alpha en utilisant la règle d'Armijo
On récupère alpha